Sequential Minimal Optimization

Serial vs Parallel Approach with Python Multithreading

Mattia Marilli

A.A. 2025/2026

Table of Content

1. Introduction
2. Theoretical Background
3. Implementation
4. Experimentation and Result

INTRODUCTION

Parallelizing SMO for SVMs

Complexity of SVMs

SVMs are powerful for binary classification but solving the quadratic optimization problem is challenging, especially due to the equality constraint that couples all variables when the problem size medium or large.

SMO and Dual Decomposition

SMO simplifies the dual SVM problem by updating two variables at a time, respecting constraints and avoiding full variable updates, though computing kernel values remains costly.

Sequential vs. Parallel Version

The parallel SMO version uses Python multithreading with the GIL disabled to compute the required kernel columns in paralellel, reducing computation time, and is compared against a sequential version to evaluate performance gains.

THEORETICAL BACKGROUND

Support Vector Machines

Find a hyperplane that maximally separates two classes.

O1

Hard-margin SVM: All points correctly classified, maximize the distance between classes.

O2

Soft-margin SVM: Allows some misclassification to handle non-separable data or outliers.

O3

Dual Problem: Optimize weights indirectly via Lagrange multipliers with equality and box constraints.

O4

Kernel Trick: Map data to higher-dimensional space implicitly to handle nonlinear separations.

Sequential Minimal Optimization

Solve the SVM dual problem by optimizing two variables at a time.

O1

Analytical updates of the two selected variables while respecting constraints.

O2

Working Set Selection: choose the Most Violating Pair to accelerate convergence.

O3

Bias term updated after each iteration to maintain correctness.

O4

Convergence guaranteed: convex objective, each step improves or maintains solution, finite extreme points.

SEQUENTIAL IMPLEMENTATION

SMO: Core Workflow

Initialization

Set Lagrange multipliers, bias, support vectors, and error cache. Prepare the model for iterative optimization, ensuring all variables start in a feasible state.

MVP Selection & Update

At each iteration, select the Most Violating Pair (the two multipliers that most violate optimality), compute only their kernel columns on the fly, and update the multipliers and bias analytically while respecting box and equality constraints.

Error Management & Convergence

Incrementally update the error cache for all training points after each update. Repeat the process until stopping criteria are met, either maximum iterations or when KKT conditions are sufficiently satisfied.

Establishing a Sequential Baseline

NumPy Parallelism

Many NumPy routines run in compiled C code and may use multiple threads, so even “single-threaded” scripts can execute some operations in parallel, masking true sequential performance.

Pure Python Implementation

Use explicit Python loops for kernel computations and avoid vectorized operations. This ensures that the sequential version reflects the performance of a genuinely single-threaded algorithm.

Reliable Baseline for Comparison

This approach allows meaningful measurement of execution time and provides a proper reference point to evaluate performance gains from parallelization.

PARALLEL IMPLEMENTATION

Disabling the GIL in Python 3.13

The GIL Limitation

The Global Interpreter Lock (GIL) prevents multiple threads from executing Python bytecode simultaneously. While it ensures thread safety, it effectively serializes CPU-bound operations.

GIL-Free Python via Custom Compilation

Pre-built Python binaries do not allow disabling the GIL. To enable true parallel execution, Python 3.13 must be compiled from source with the --disable-gil flag.

Benchmark and Usage

GIL-free execution allows parallel CPU-bound computations like kernel matrix evaluation. Scripts can be run with PYTHON_GIL=0 python yournogilscript.py to verify performance improvements.

Multithreaded Kernel Computation

Thread Pool Execution

The sequential SMO is extended by distributing kernel column computation across multiple threads using a ThreadPoolExecutor. Each thread evaluates a subset of entries independently, avoiding conflicts.

Parallel Kernel Computation

For each Most Violating Pair, the corresponding kernel columns are computed in parallel. Each thread processes a contiguous block of training samples, ensuring efficient workload distribution and near-linear speedup.

Incremental Updates & Correctness

Once the kernel columns for the MVP are computed, the error cache is updated sequentially. Threads write to distinct indices, eliminating the need for locks and ensuring data integrity.

EXPERIMENTATION AND RESULTS

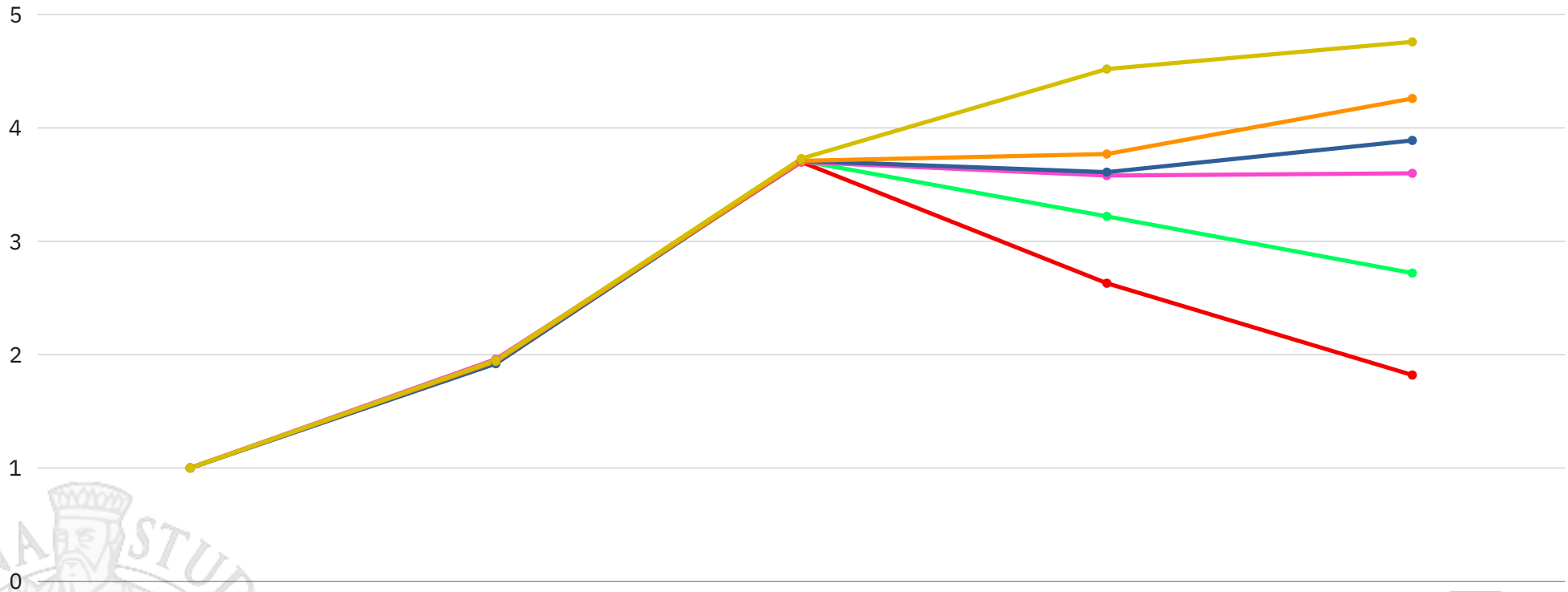
Test Setup

- **Synthetic datasets** were generated using make_classification with **3000 samples** and a varying **number of features from 20 to 600**. The datasets contained balanced binary classes and standardized features to ensure consistency across experiments.
- Each dataset was split into training and testing sets using an 80/20 ratio, and features were normalized with StandardScaler. The **RBF kernel** parameter γ was set to the inverse of the number of features, while the regularization parameter C was fixed at 1.
- The S**VM was trained multiple times** for each dataset using different thread counts, specifically 1, 2, 4, 8, and 16 threads. Execution time was measured separately for the computation of kernel columns, which represents the main computational cost. **Multiple runs** were averaged to reduce performance fluctuations.
- All experiments were performed on an **Apple M1 processor with 8 cores**, consisting of 4 high-performance and 4 high-efficiency cores.

Speedup

The following graph illustrates the speedup achieved with increasing numbers of threads for different feature counts.

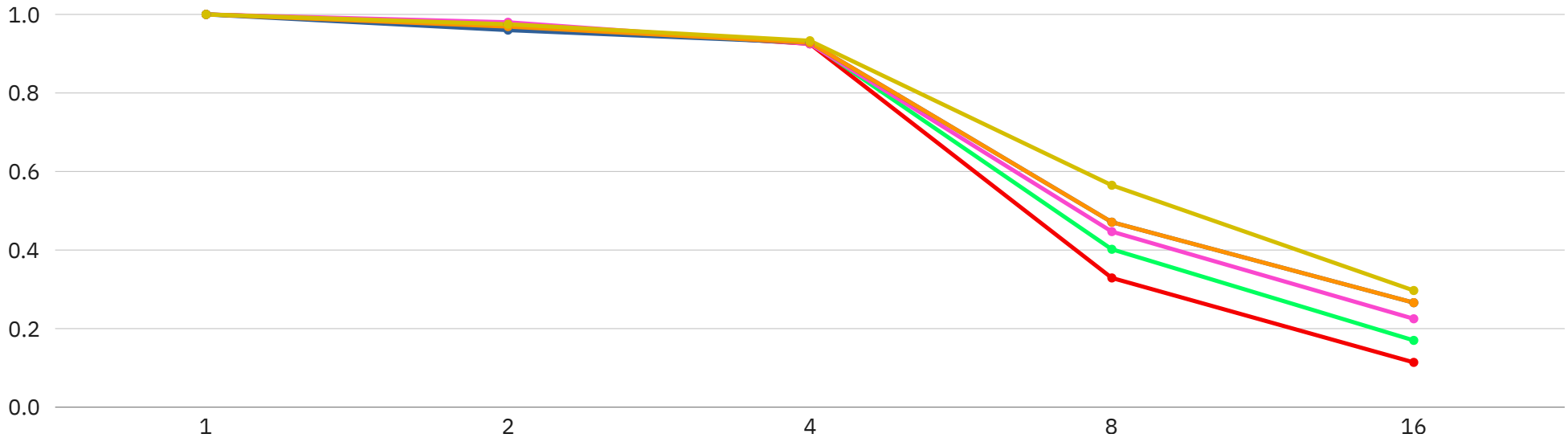
It shows the relationship between the number of threads (from 1 to 16) and the corresponding speedup for various numbers of features (from 20 to 600).



Efficiency

The following graph illustrates the efficiency achieved with increasing numbers of threads for different features counts.

It shows the relationship between the number of threads (from 1 to 16) and the corresponding efficiency for various numbers of features (from 20 to 600).



Results Analysis

Performance Scaling

The parallel SMO achieves near-linear speedup up to four threads, corresponding to the high-performance cores of the Apple M1. Beyond that, efficiency gradually decreases because of thread scheduling and heterogeneous core behavior

Dataset Size Impact

As the number of features increases, datasets show better scalability. Larger workloads allow the overhead of thread management to be amortized, while smaller datasets saturate earlier due to limited per-thread computation.

Accuracy Preservation

Across all configurations, classification accuracy remains identical to the sequential baseline, confirming that parallel kernel computation preserves correctness and numerical stability.

Results Analysis

Speedup Trends

The speedup grows almost linearly up to four threads for all datasets. With larger feature counts, gains extend further, reaching values above 4.5× with 16 threads.

Parallel Efficiency

Efficiency remains above 90% up to four threads, confirming optimal parallel utilization. At higher thread counts, efficiency decreases progressively because of synchronization overhead and CPU heterogeneity.

Overall Assessment

The results confirm that parallelizing kernel computation within SMO provides substantial performance benefits. It achieves strong scalability on multi-core systems while maintaining identical model accuracy.

Thank you!

Project Repository

Click Here